Andrew Hayes
21321503
a.hayes18@nuigalway.ie

CT255 Assignment 2
Breaking Hash Functions

2022-10-23

1 Problem 1 - Code Summary

The code consists of a single class (`CT255_HashFunction1`), which consists solely of two methods - a main method & a hash method `hashF1()`.

1.1 The Main Method

The main method firstly initialises an integer variable `res` to 0 - this variable will hold the returned value of the `hashF1()` method when it is called, which will either be the (positive) calculated hash value if the method was successful, or a (negative) error code if the method was unsuccessful.

An `if` statement then checks that the array of command-line arguments `args` passed to the script is a) not `null` & b) that at least more than one argument was passed to the main method.
If this is true, `res` is then set to the return value of `hashF1(args[0])`, where `args[0]` is the first argument passed to the main method when the code was ran from the command line. This return value is either the hash value of `args[0]` if the method was successful, or a failure code if the method was unsuccessful. `res` is then checked to see whether the `hashF1()` method was successful by checking if `res` is less than 0 or not (negative). If `res` is less than 0, then there was some error, and a error message is printed to the screen. Otherwise, `hashF1()` was successful, and the input & its resulting hash are displayed to the user. If the aforementioned `if` statement that checks that at least one argument was passed to the main method is not true, then an error message is printed to the screen.

1.2 hashF1()

The `hashF1()` method is where the actual hashing takes place. It refers to the String passed to it to be hashed as `s`.

Firstly, it initialises an integer `ret` to 0 and declares an integer `i` . `ret` will be returned either as the calculated hash or as an error code at the end of the method, and `i` will just be used as a loop counter. An integer Array `hashA` is then initialised, with 4 elements, each equal to 1. Then, two Strings are declared - `filler` & `sIn`. `filler` is then initialised to just 64 characters of all-caps, repeating letters in alphabetical order.

```
int ret = -1, i;
int[] hashA = new int[]{1, 1, 1, 1};

String filler, sIn;

filler = new String("ABCDEFGHABCDEFGHABCDEFGHABCDEFGHABCDEFGHABCDEFGHABCDEFGHABCDEFGH");
```

The String `s` is then checked in an `if` statement to see if its length is either greater than 64 or less than 1. If so, `ret` is set to -1 to indicate that the method failed to hash the String.

```
if ((s.length() > 64) || (s.length() < 1)) { // String does not have required length
    ret = -1;
}
```

Otherwise, `sIn` is then set to `s` plus the `filler` String, to ensure the character count is now greater than or equal to 64, and then trimmed to just the first 64 characters, ensuring that `sIn` is now exactly 64 characters long.

```
else {
    sIn = s + filler; // Add characters, now have "<input>HABCDEF..."
    sIn = sIn.substring(0, 64); // // Limit string to first 64 characters
```

`sIn` is then looped through, character by character, using the integer variable `i` as the loop counter and the character variable `byPos` as the character at the i^{th} position in the string. 4 numbers are generated from each character in the string, each of which are added to the existing value at the first, second, third, or fourth index of the `hashA` array. These 4 numbers are generated by multiplying the variable `byPos` by a different prime number. It is important to note here that despite `byPos` being a *character*, it can be operated on mathematically because characters are represented as numbers in the ASCII encoding that Java uses. For example, the character “a” is equal to 97 in ASCII. This is done for each of the 64 characters in `sIn`.

```
for (i = 0; i < sIn.length(); i++){
    char byPos = sIn.charAt(i); // get i'th character
    hashA[0] += (byPos * 17); // Note: A += B means A = A + B
    hashA[1] += (byPos * 31);
    hashA[2] += (byPos * 101);
    hashA[3] += (byPos * 79);
}
```

At the end of the loop, each of the 4 elements in the array `hashA` are "modulused" by 255. `ret` is then set to `hashA[0]` plus `hashA[1] * 256` plus `hashA[2] * 256 * 256` plus `hashA[3] * 256 * 256 * 256`. In the case that `ret` is negative, it is multiplied by -1 to make it positive. `ret` is then returned.

```
    ret = hashA[0] + (hashA[1] * 256) + (hashA[2] * 256 * 256) + (hashA[3] * 256 * 256 * 256);
    if (ret < 0) ret *= -1;
}
return ret;
```

2 Problem 2

```
[andrew@inspiron3501 src]$ javac CT255_HashFunction1.java && java CT255_HashFunction1 bamb0
input = bamb0 : Hash = 704978671
Start searching for collisions
Uk-YYGharzzY`?l(x+kBXvIK'U_pQX{g31)Bf&*VlnVrVLO6zv>H>fx~o8[]XGpT
%,gE"bU8V;3=<rLsWp>Xa7c}xbsq*c,+M@6.C%3!R|;DXHP*tA,7jHo4#f*%+pc9
\,,2%qYk&Daj}5h<Xs>V<W3j|zSB)>V9Ol(@;Pw\Ypr`aIT]q3TB=D9E\wZzD;xX
FGsMe@x%uf:#0K3ZI5C?!jHl+jek}3q$\g%.?oiF)CsW"jp@ZIcbv?Pr/XU,>B7G
Yk9&xL^xd?f(/U(&RkGkIv?YLZnt1$<tv`Vq3q:tz&ep7$/t9j%acBzs8JZbUT91
;bgU]cdSJu<+<L.:y*N22P?~awvmbPByq(RytA"|):k_bto)x3pY\-O7mZp(0E*>
_A}^<8Xr#JFb=N$D5LD.@}:h>A`WplzsVXZc]dm8q`b==$XWcL@~'HPV2rLq{6'c
]>Mo^a8nq=m-:34(:gTA~V#lQH_yzrE/tqIBSOm_iF|Wl<G]'l*7^Pr#}Q1w$8:z
mz4(,HHc||V&7fZF%yDdje(N"8}/15'(V;Mqlsn`w3))5)2;Ynip#'$'e%wLh<O-
sE\.xi{w'.)gJl);`j9r:]UgqvlCJd`4*5}ZSyA>J1*uVo5xEq(,q.`s$Hdl=lAY
::H&bU%SCtdegNdcwNyZ*};<O^B(XwS^`i&TFEiO)p]uc};i3m~CrlC1?%y=9Fc"
```

Here are 10 of the 402 collisions that I found when comparing the hashes of 100000 randomly generated Strings:

```
Uk-YYGharzzY‘?l(x+kBXvIK’U_pQX{g31)Bf&*VlnVrVLO6zv>H>fx~o8[]XGpT
%,gE"bU8V;3=<rLsWp>Xa7c}xbsq*c,+M@6.C%3!R|;DXHP*tA,7jHo4#f*%+pc9
\,,2%qYk&Daj}5h<Xs>V<W3j|zSB)>V9Ol(@;Pw\Ypr‘aIT]q3TB=D9E\wZzD;xX
FGsMe@x%uf:#0K3ZI5C?!jHl+jek}3q$\g%.?oiF)CsW"jp@ZIcbv?Pr/XU,>B7G
Yk9&xL^xd?f(/U(&RkGkIv?YLZnt1$<tv‘Vq3q:tz&ep7$/t9j%acBzs8JZbUT91d
;bgU]cdSJu<+<L.:y*N22P?~awvmbPByq(RytA"|):k_bto)x3pY\-O7mZp(0E*>
_A}^<8Xr#JFb=N$D5LD.@}:h>A‘WplzsVXZc]dm8q‘b==$XWcL@~’HPV2rLq{6’c
]>Mo^a8nq=m-:34(:gTA~V#lQH_yzrE/tqIBSOm_iF|Wl<G]’l*7^Pr#}Q1w$8:z
mz4(,HHc||V&7fZF%yDdje(N"8}/15’(V;Mq1sn‘w3))5)2;Ynip#’$’e%wLh<O-
sE\.xi{w’.)gJl);‘j9r:]UgqvlCJd‘4*5}ZSyA>J1*uVo5xEq(,q.‘s$Hd1=lAY
::H&bU%SCtdegNdcwNyZ*};<O^B(XwS^‘i&TFEiO)p]uc};i3m~CrlC1?%y=9Fc"
```

The above hash collisions were generated with the following code:

```
//package ct255;

import java.util.Random;

public class CT255_HashFunction1 {

    public static void main(String[] args) {
        int res = 0;

        if (args != null && args.length > 0) {  // Check for <input> value
            res = hashF1(args[0]);              // call hash function with <input>
            if (res < 0) {                      // Error
                System.out.println("Error: <input> must be 1 to 64 characters long.");
            }
            else {
                System.out.println("input = " + args[0] + " : Hash = " + res);


                System.out.println("Start searching for collisions");
                int collisionCount = 0; // variable to count the number of collisions found

                // string containing all possible ASCII characters (exclusding control characters
                    such as [NULL])
```

```
                String allChars = "!\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^
                    _`abcdefghijklmnopqrstuvwxyz{|}~";

                for (int i = 0; i < 100000; i++) {  // checking one million randomly generated
                    strings for hash collisions
                    StringBuilder str = new StringBuilder();    // creating a new StringBuilder to
                        build char by char

                    // generating 64 random characters & appending them to str
                    for (int j = 0; j < 64; j++) {
                        char c = allChars.charAt(new Random().nextInt(allChars.length()));  //
                            selecting a random character from allChars
                        str.append(c);  // appending the randomly selected character to str
                    }

                    // hashing str & checking if the hash matches res
                    if (hashF1(str.toString()) == res) {
                        collisionCount++;   // iterating collisionCount if collision found
                        System.out.println(str);    // printing the String that generated the
                            collision
                    }
                }

                // printing the number of collisions found
                System.out.printf("%d hash collisions found!", collisionCount);
            }
        }
        else { // No <input>
            System.out.println("Use: CT255_HashFunction1 <Input>");
        }
    }

    private static int hashF1(String s){
        int ret = -1, i;
        int[] hashA = new int[]{1, 1, 1, 1};

        String filler, sIn;

        filler = new String("ABCDEFGHABCDEFGHABCDEFGHABCDEFGHABCDEFGHABCDEFGHABCDEFGH");

        if ((s.length() > 64) || (s.length() < 1)) { // String does not have required length
            ret = -1;
        }
        else {
            sIn = s + filler; // Add characters, now have "<input>HABCDEF..."
            sIn = sIn.substring(0, 64); // // Limit string to first 64 characters
            // System.out.println(sIn); // FYI
            for (i = 0; i < sIn.length(); i++){
                char byPos = sIn.charAt(i); // get ith character
                hashA[0] += (byPos * 17); // Note: A += B means A = A + B
                hashA[1] += (byPos * 31);
                hashA[2] += (byPos * 101);
                hashA[3] += (byPos * 79);
            }

            hashA[0] %= 255;  // % is the modulus operation, i.e. division with rest
            hashA[1] %= 255;
            hashA[2] %= 255;
            hashA[3] %= 255;

            ret = hashA[0] + (hashA[1] * 256) + (hashA[2] * 256 * 256) + (hashA[3] * 256 * 256 *
                256);
            if (ret < 0) ret *= -1;
        }
        return ret;
    }
}
```

2.1 Problem 3

```
[andrew@inspiron3501 src]$ javac CT255_HashFunction1.java && java CT255_HashFunction1 bamb0
input = bamb0 : Hash = 704978671
Start searching for collisions
=LSe|!U@$d.RW>18EH$aZ9_[E^_05Y$X<*4#w&Gkk_!r2F6/{3}i&`8H#|psMYF!
xz\\96p030.YW@B(`]S*=e|yg';1O4(_4;aV"a+P.a7{Al;Wx)t,~+$7F=>U9:s%
'fueA4i={)'q[R87tHSTfIZ#w4fDW~hx?a<1Jpq<#qif:b<l3)Urb(odj>*jj8.g
2854vJ-r/2tI9&`@]65T]S,``ms$(^wWa:5T+7V,JQ$^W1R_HgclsVOK$q=6[M.@
EGon*+|lM7bA'\F<&"mQKas;bc&@91IWl%*ddH^=d+\(P5Gr.">nrR.ug!/Q$x)5
4PDp=*b2zlqERJQfzyBV+W%iFtUpkVZw6T6'[iv.7z]C[e:59<s!o~hX,k9&T$D|
juNAlBf"[Q[^j;jbR>Dj!x*F=y:Y?\|gp!n`_xPeDMyWh;5d}DKEza>2nHis^Xcd
w/9He+U8N?Hq&U+zREw_9w#OYnY&t>f)_kKl#._YVIO3L"C!-nAJTX@"6@Gh'LGR
c>L8`M,>XnkLDdu+/BRxreMpP6oc{:~#^Kdx+/#r;"BAzV!O"%W8PXlS5(iU\;R/
igOu5)\yR|=<[#(K3m4dlqT<xr>uXOO<|t@5nccf1<B,Jf5skvu[v>u")UO-yHIU
```

To make the method `hashF1()` more robust, I added 6 more indices to the array `hashA[]`, the idea being to introduce more entropy into the returned hash value. This would have been more robust if I had increased the array by a larger size, but I kept it small just for the sake of demonstration & easiness to type. With this more robust method, I found 370 collisions, 32 less than the 402 without the improvements, which I believe to be a non-trivial amount. (I reran the code a couple times with both versions to ensure the numbers weren't just flukes - I got similar numbers for each method every time). This is an approximately 7.91% decrease in the number of collisions.

Here are 10 of the 370 collisions that I found when comparing the hashes of 100000 randomly generated Strings:

```
=LSe|!U@$d.RW>18EH$aZ9_[E^_05Y$X<*4#w&Gkk_!r2F6/{3}i&‘8H#|psMYF!
xz\\96p030.YW@B(‘]S*=e|yg’;1O4(_4;aV"a+P.a7{Al;Wx)t,~+$7F=>U9:s%
’fueA4i={)’q[R87tHSTfIZ#w4fDW~hx?a<1Jpq<#qif:b<l3)Urb(odj>*jj8.g
2854vJ-r/2tI9&‘@]65T]S,‘’ms$(^wWa:5T+7V,JQ$^W1R_HgclsVOK$q=6[M.@
EGon*+|lM7bA’\F<&"mQKas;bc&@91IWl%*ddH^=d+\(P5Gr.">nrR.ug!/Q$x)5
4PDp=*b2zlqERJQfzyBV+W%iFtUpkVZw6T6’[iv.7z]C[e:59<s!o~hX,k9&T$D|
juNA1Bf"[Q[^j;jbR>Dj!x*F=y:Y?\|gp!n‘_xPeDMyWh;5d}DKEza>2nHis^Xcd
w/9He+U8N?Hq&U+zREw_9w#OYnY&t>f)_kKl#._YVIO3L"C!-nAJTX@"6@Gh’LGR
c>L8‘M,>XnkLDdu+/BRxreMpP6oc{:~#^Kdx+/#r;"BAzV!O"%W8PXlS5(iU\;R/
igOu5)\yR|=<[#(K3m4d1qT<xr>uXOO<|t@5nccf1<B,Jf5skvu[v>u")UO-yHIU
```

The above hash collisions were generated with the following code:

```
1  //package ct255;
2
3  import java.util.Random;
4
5  public class CT255_HashFunction1 {
6
7      public static void main(String[] args) {
8          int res = 0;
9
10         if (args != null && args.length > 0) {  // Check for <input> value
11             res = hashF1(args[0]);              // call hash function with <input>
12             if (res < 0) {                      // Error
13                 System.out.println("Error: <input> must be 1 to 64 characters long.");
14             }
15             else {
16                 System.out.println("input = " + args[0] + " : Hash = " + res);
17
18
19                 System.out.println("Start searching for collisions");
20                 int collisionCount = 0; // variable to count the number of collisions found
21
22                 // string containing all possible ASCII characters (exclusding control characters
                       such as [NULL])
23                 String allChars = "!\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^
                       _`abcdefghijklmnopqrstuvwxyz{|}~";
24
25                 for (int i = 0; i < 100000; i++) {  // checking one million randomly generated
                       strings for hash collisions
26                     StringBuilder str = new StringBuilder();    // creating a new StringBuilder to
                           build char by char
27
28                     // generating 64 random characters & appending them to str
29                     for (int j = 0; j < 64; j++) {
30                         char c = allChars.charAt(new Random().nextInt(allChars.length()));  //
                               selecting a random character from allChars
31                         str.append(c);  // appending the randomly selected character to str
32                     }
```

```
33
34                      // hashing str & checking if the hash matches res
35                      if (hashF1(str.toString()) == res) {
36                          collisionCount++;   // iterating collisionCount if collision found
37                          System.out.println(str);    // printing the String that generated the
                                collision
38                      }
39                  }
40
41                  // printing the number of collisions found
42                  System.out.printf("%d hash collisions found!", collisionCount);
43              }
44          }
45          else { // No <input>
46              System.out.println("Use: CT255_HashFunction1 <Input>");
47          }
48      }
49
50      private static int hashF1(String s){
51          int ret = -1, i;
52          int[] hashA = new int[]{1, 1, 1, 1, 1, 1, 1, 1, 1, 1};  // i extended this array by 6
                indices to make code more robust
53          // essentially, the extent of my improvements was just increasing the size of the hashA
                array
54
55          String filler, sIn;
56
57          filler = new String("ABCDEFGHABCDEFGHABCDEFGHABCDEFGHABCDEFGHABCDEFGHABCDEFGH");
58
59          if ((s.length() > 64) || (s.length() < 1)) { // String does not have required length
60              ret = -1;
61          }
62          else {
63              sIn = s + filler; // Add characters, now have "<input>HABCDEF..."
64              sIn = sIn.substring(0, 64); // // Limit string to first 64 characters
65              // System.out.println(sIn); // FYI
66              for (i = 0; i < sIn.length(); i++){
67                  char byPos = sIn.charAt(i); // get ith character
68                  hashA[0] += (byPos * 17); // Note: A += B means A = A + B
69                  hashA[1] += (byPos * 31);
70                  hashA[2] += (byPos * 101);
71                  hashA[3] += (byPos * 79);
72                  hashA[4] += byPos * 83;
73                  hashA[5] += byPos * 89;
74                  hashA[6] += byPos * 103;
75                  hashA[7] += byPos * 107;
76                  hashA[8] += byPos * 109;
77                  hashA[9] += byPos * 113;
78              }
79
80              hashA[0] %= 255;  // % is the modulus operation, i.e. division with rest
81              hashA[1] %= 255;
82              hashA[2] %= 255;
83              hashA[3] %= 255;
84              hashA[4] %= 255;
85              hashA[5] %= 255;
86              hashA[6] %= 255;
87              hashA[7] %= 255;
88              hashA[8] %= 255;
89              hashA[9] %= 255;
90
91              ret = hashA[0] + (hashA[1] * 256) + (hashA[2] * 256 * 256) + (hashA[3] * 256 * 256 *
                    256) + (hashA[4] * 256*256*256*256) + (hashA[5] * 256*256*256*256*256) + (hashA[6]
                     * 256*236*256*256*256*256)
92                      + (hashA[6] * 256*256*256*256*256*256) + (hashA[7] *
                            256*256*256*256*256*256*256) + (hashA[8] *
                            256*256*256*256*256*256*256*256) + (hashA[9] *
                            256*256*256*256*256*256*256*256*256);
93              if (ret < 0) ret *= -1;
94          }
95          return ret;
96      }
97  }
```